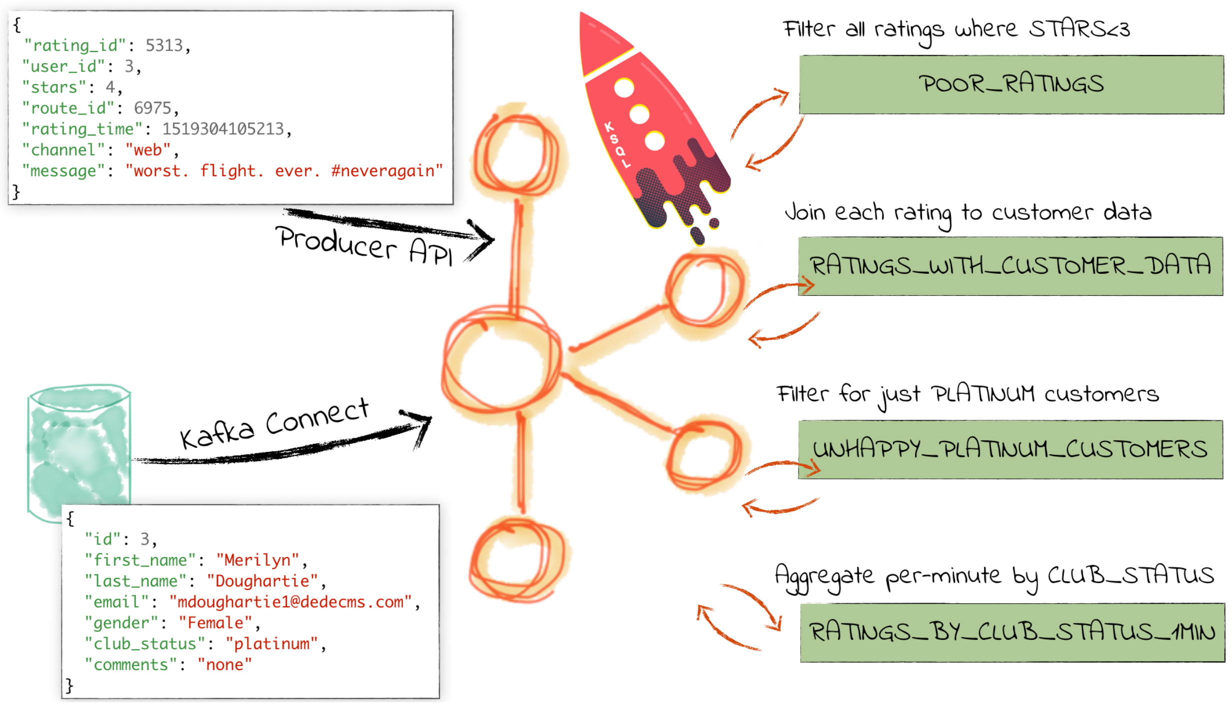
Workshop: Real-time SQL Stream Processing at Scale with Apache Kafka and KSQL (Mac/Linux)

Table of Contents

1. Introduction

> KSQL is the streaming SQL engine for Apache Kafka. Using just SQL, it is possible for developers to build powerful stream processing applications. This workshop will show practical examples of KSQL:
>
> - Filtering streams of data
> - Joining live events with reference data (e.g., from a database)
> - Stateful aggregations
> - Convert streams from JSON to AVRO



```
{
 "rating_id": 5313,
 "user_id": 3,
 "stars": 4,
 "route_id": 6975,
 "rating_time": 1519304105213,
 "channel": "web",
 "message": "worst. flight. ever. #neveragain"
}
```

```
{
  "id": 3,
  "first_name": "Merilyn",
  "last_name": "Doughartie",
  "email": "mdoughartie1@dedecms.com",
  "gender": "Female",
  "club_status": "platinum",
  "comments": "none"
}
```

Producer API
Kafka Connect
KSQL
Filter all ratings where STARS<3
POOR_RATINGS
Join each rating to customer data
RATINGS_WITH_CUSTOMER_DATA
Filter for just PLATINUM customers
UNHAPPY_PLATINUM_CUSTOMERS
Aggregate per-minute by CLUB_STATUS
RATINGS_BY_CLUB_STATUS_1MIN

You can find all of the KSQL commands used during this workshop in the `ksql-workshop.sql` file.

The slides that go with it can be found here.

Note	For Windows instructions, please see ksql-workshop-windows.adoc

Don't forget to check out the `#ksql` channel on our Community Slack group

——*Robin Moffatt @rmoff*

2. Prerequisites

Important	These **MUST** be completed before the workshop!

Please see pre-requisites.adoc

3. Start Confluent Platform

Note	Make sure that Docker has at least 8GB of memory available (check with `docker system info | grep Memory`)

Note	For Windows instructions, please see ksql-workshop-windows.adoc

Change the working directory:

```
cd demo-scene/ksql-workshop
```

Start Confluent Platform:

```
docker-compose up -d
docker-compose logs -f kafka|grep "INFO Kafka version"
```

Once you see output, then it means Kafka is running and you can proceed

```
$ docker-compose logs -f kafka|grep "INFO Kafka version"
kafka                | [2019-09-20 17:54:24,968] INFO Kafka version: 5.3.1-
```

Press **Ctrl-C** twice to exit the `docker-compose logs` command

Run `docker-compose ps` to confirm that all components are running:

```
$ docker-compose ps
      Name                     Command                State
---------------------------------------------------------------------------
connect-debezium   /docker-entrypoint.sh start      Up      0.0.0.0:8083->
control-center     bash -c echo "Waiting two  ...   Up      0.0.0.0:9021->
datagen-ratings    bash -c echo Waiting for K ...   Up
elasticsearch      /usr/local/bin/docker-entr ...   Up      0.0.0.0:9200->
kafka              /etc/confluent/docker/run        Up      0.0.0.0:9092->
kafka-connect      /etc/confluent/docker/run        Up      0.0.0.0:18083-
kafkacat           sleep infinity                   Up
kibana             /usr/local/bin/kibana-docker     Up      0.0.0.0:5601->
ksql-cli           /bin/sh                          Up
ksql-server        /etc/confluent/docker/run        Up      0.0.0.0:8088->
mysql              docker-entrypoint.sh mysqld      Up      0.0.0.0:3306->
nginx              nginx -g daemon off;             Up      80/tcp, 0.0.0.
schema-registry    /etc/confluent/docker/run        Up      8081/tcp
websockets         docker-entrypoint.sh bash  ...   Up      3000/tcp
zookeeper          /etc/confluent/docker/run        Up      2181/tcp, 2888
```

Important	If any components do not show "Up" under the `State` column (e.g., they say "Exit") then you must rectify this before continuing. As a first solution, try re-issuing the `docker-compose up -d` command.

4. KSQL

KSQL can be used via the command-line interface (CLI), a graphical UI built into Confluent Control Center, or the documented REST API.

In this workshop, we will use the CLI, which if you have used Oracle's `sql*plus`, MySQL CLI, and so on will feel very familiar to you.

Launch the CLI:

```
docker-compose exec ksql-cli ksql http://ksql-server:8088
```

Make sure that you get a successful start up screen:

```
                  ===========================================
                  =        _  __ _____  ____  _             =
                  =       | |/ // ____|/ __ \| |            =
                  =       | ' /| (___ | |  | | |            =
                  =       |  <  \___ \| |  | | |            =
                  =       | . \ ____) | |__| | |____        =
                  =       |_|\_\_____/ \___\_\______|       =
                  =                                         =
                  =  Streaming SQL Engine for Apache Kafka® =
                  ===========================================

Copyright 2017-2018 Confluent Inc.

CLI v5.2.1, Server v5.2.1 located at http://ksql-server:8088

Having trouble? Type 'help' (case-insensitive) for a rundown of how things

ksql>
```

4.1. See available Kafka topics

KSQL can be used to view the topic metadata on a Kafka cluster (`SHOW TOPICS;`), as well as inspect the messages in a topic (`PRINT <topic>;`).

```
ksql> SHOW TOPICS;

 Kafka Topic                     | Registered | Partitions | Partition Rep
--------------------------------------------------------------------------
 _confluent-metrics              | false      | 12         | 1
 _schemas                        | false      | 1          | 1
 ratings                         | false      | 1          | 1
[...]
--------------------------------------------------------------------------
ksql>
```

4.2. Inspect a topic contents - Ratings

The event stream driving this example is a simulated stream of events purporting to show the ratings left by users on a website, with data elements including the device type that they used, the star rating, and a message associated with the rating.

Using the `PRINT` command we can easily see column names and values within a topic's messages. Kafka messages consist of a timestamp, key, and message (payload), which are all shown in the `PRINT` output.

Tip	Note that we don't need to know the format of the data; KSQL introspects the data and understands how to deserialise it.

```
ksql> PRINT 'ratings';
Format:AVRO
22/02/18 12:55:04 GMT, 5312, {"rating_id": 5312, "user_id": 4, "stars": 4,
22/02/18 12:55:05 GMT, 5313, {"rating_id": 5313, "user_id": 3, "stars": 4,
```

Press Ctrl-C to cancel and return to the KSQL prompt.

Note

Optional: You can also inspect the Kafka topic with a tool such as `kafkacat`
Kafka messages, but it also includes detailed metadata.

```
docker-compose exec kafkacat \
          kafkacat -b kafka:29092 -C -K: \
          -f '\nKey (%K bytes): %k\t\nValue (%S bytes): %s\n\Part
          -t ratings -o end
```

Note that the data is in Avro, so you will see lots of special characters in the

Press Ctrl-C to cancel and return to the command prompt.

5. Querying and filtering streams of data

Having inspected the topics and contents of them, let's get into some SQL now. The first step in KSQL is to register the source topic with KSQL.

5.1. Register the ratings topic

By registering a topic with KSQL, we declare its schema and properties.

The inbound event stream of ratings data is a `STREAM` —later we will talk about `TABLE` , but for now, we just need a simple `CREATE STREAM` with the appropriate values in the `WITH` clause:

```
ksql> CREATE STREAM ratings WITH (KAFKA_TOPIC='ratings', VALUE_FORMAT='AVR

 Message
---------------
```

```
 Stream created
---------------
```

Note	Here we’re using data in Avro format, but you can also read and write JSON or CSV data with KSQL. If you use JSON or CSV you have to specify the schema when you create the stream.

5.2. Describe ratings stream

You’ll notice that in the above `CREATE STREAM` statement we didn’t specify any of the column names. That’s because the data is in Avro format, and the Confluent Schema Registry supplies the actual schema details. You can use `DESCRIBE` to examine an object’s columns:

```
ksql> DESCRIBE ratings;
Name                 : RATINGS
 Field       | Type
-----------------------------------------
 ROWTIME     | BIGINT           (system)
 ROWKEY      | VARCHAR(STRING)  (system)
 RATING_ID   | BIGINT
 USER_ID     | INTEGER
 STARS       | INTEGER
 ROUTE_ID    | INTEGER
 RATING_TIME | BIGINT
 CHANNEL     | VARCHAR(STRING)
 MESSAGE     | VARCHAR(STRING)
-----------------------------------------
For runtime statistics and query details run: DESCRIBE EXTENDED <Stream,Ta
ksql>
```

Note the presence of a couple of `(system)` columns here. `ROWTIME` is the timestamp of the Kafka message—important for when we do time-based aggregations later—and `ROWKEY` is the key of the Kafka message.

5.3. Querying data in KSQL

Let’s run our first SQL. As anyone familar with SQL knows, `SELECT *` will return all columns from a given object. So let’s try it!

```
ksql> SELECT * FROM ratings;
1529501380124 | 6229 | 6229 | 17 | 2 | 3957 | 1529501380124 | iOS-test | w
1529501380197 | 6230 | 6230 | 14 | 2 | 2638 | 1529501380197 | iOS | your t
1529501380641 | 6231 | 6231 | 12 | 1 | 9870 | 1529501380641 | iOS-test | (
[…]
```

You’ll notice that the data keeps on coming. That is because KSQL is fundamentally a *streaming engine*, and the queries that you run are *continuous queries*. Having previously set the offset to `earliest` KSQL is showing us the **past** (data from the beginning of the topic), the **present** (data now arriving in the topic), and the **future** (all new data that arrives in the topic from now on).

Press Ctrl-C to cancel the query and return to the KSQL command prompt.

To inspect a finite set of data, you can use the `LIMIT` clause. Try it out now:

```
ksql> SELECT * FROM ratings LIMIT 5;
1529499830648 | 1 | 1 | 8 | 1 | 7562 | 1529499829398 | ios | more peanuts
1529499830972 | 2 | 2 | 5 | 4 | 54 | 1529499830972 | iOS | your team here
1529499831203 | 3 | 3 | 16 | 1 | 9809 | 1529499831203 | web | airport refu
1529499831521 | 4 | 4 | 5 | 1 | 7691 | 1529499831521 | web | thank you for
1529499831814 | 5 | 5 | 19 | 3 | 389 | 1529499831814 | ios | thank you for
Limit Reached
Query terminated
ksql>
```

5.4. Filtering streams of data in KSQL

Since KSQL is heavily based on SQL, you can do many of the standard SQL things you’d expect to be able to do, including predicates and selection of specific columns:

```
ksql> SELECT USER_ID, STARS, CHANNEL, MESSAGE FROM ratings WHERE STARS <3
3 | 2 | iOS | your team here rocks!
2 | 1 | iOS | worst. flight. ever. #neveragain
15 | 2 | iOS | worst. flight. ever. #neveragain
Limit Reached
Query terminated
ksql>
```

5.5. KSQL offsets

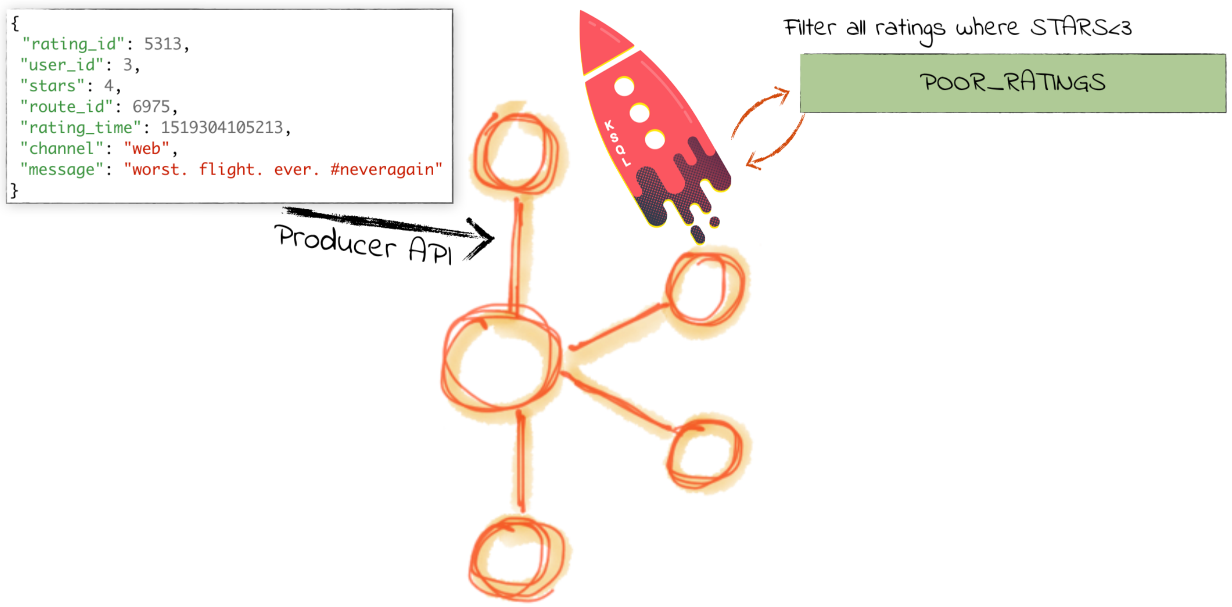
Since Apache Kafka persists data, it is possible to use KSQL to query and process data from the past, as well as new events that arrive on the topic.

To tell KSQL to process from beginning of topic run `SET 'auto.offset.reset' = 'earliest';`

Run this now, so that future processing includes all existing data.

```
ksql> SET 'auto.offset.reset' = 'earliest';
Successfully changed local property 'auto.offset.reset' from 'null' to 'ea
```

6. Creating a Kafka topic populated by a filtered stream



```
{
  "rating_id": 5313,
  "user_id": 3,
  "stars": 4,
  "route_id": 6975,
  "rating_time": 1519304105213,
  "channel": "web",
  "message": "worst. flight. ever. #neveragain"
}
```

Let’s take the poor ratings from people with iOS devices, and create a new stream from them!

```
ksql> CREATE STREAM POOR_RATINGS AS SELECT * FROM ratings WHERE STARS <3 A

 Message
-------------------------------
 Stream created and running
-------------------------------
```

What this does is set a KSQL continuous query running that processes messages on the source `ratings` topic to:

- applies the predicates (`STARS<3 AND CHANNEL='iOS'``)
- selects just the specified columns
    - If you wanted to take all columns from the source stream, you would simply use `SELECT *`

Each processed message is written to a new Kafka topic. Remember, this is a *continuous query*, so every single source message—past, present, and future—will be processed with low-latency in this way. If we only want to process *new* messages and not existing ones, we would configure `SET 'auto.offset.reset' = 'latest';` .

This method of creating derived topics is frequently referred to by the acronym of the statement— `CSAS` (→ `CREATE STREAM … AS SELECT`).

6.1. Inspect the derived stream

Using `DESCRIBE` we can see that the new stream has the same columns as the source one.

```
ksql> DESCRIBE POOR_RATINGS;
Name                 : POOR_RATINGS
 Field       | Type
-----------------------------------------
 ROWTIME     | BIGINT           (system)
 ROWKEY      | VARCHAR(STRING)  (system)
 RATING_ID   | BIGINT
 USER_ID     | INTEGER
 STARS       | INTEGER
 ROUTE_ID    | INTEGER
 RATING_TIME | BIGINT
 CHANNEL     | VARCHAR(STRING)
 MESSAGE     | VARCHAR(STRING)
-----------------------------------------
For runtime statistics and query details run: DESCRIBE EXTENDED <Stream,Ta
ksql>
```

Additional information about the derived stream is available with the `DESCRIBE EXTENDED` command:

```
ksql> DESCRIBE EXTENDED POOR_RATINGS;
Name                 : POOR_RATINGS
Type                 : STREAM
Key field            :
Key format           : STRING
Timestamp field      : Not set - using <ROWTIME>
Value format         : AVRO
Kafka topic          : POOR_RATINGS (partitions: 4, replication: 1)

 Field       | Type
-----------------------------------------
 ROWTIME     | BIGINT           (system)
 ROWKEY      | VARCHAR(STRING)  (system)
 RATING_ID   | BIGINT
 USER_ID     | INTEGER
 STARS       | INTEGER
 ROUTE_ID    | INTEGER
 RATING_TIME | BIGINT
 CHANNEL     | VARCHAR(STRING)
 MESSAGE     | VARCHAR(STRING)
-----------------------------------------

Queries that write into this STREAM
-----------------------------------
CSAS_POOR_RATINGS_0 : CREATE STREAM POOR_RATINGS AS SELECT * FROM ratings

For query topology and execution plan please run: EXPLAIN <QueryId>

Local runtime statistics
```

```
------------------------
messages-per-sec:     10.04   total-messages:       998     last-message:
 failed-messages:         0 failed-messages-per-sec:         0      last-f
(Statistics of the local KSQL server interaction with the Kafka topic POOR
ksql>
```

Note the **runtime statistics** above. If you re-run the `DESCRIBE EXTENDED` command you'll see these values increasing.

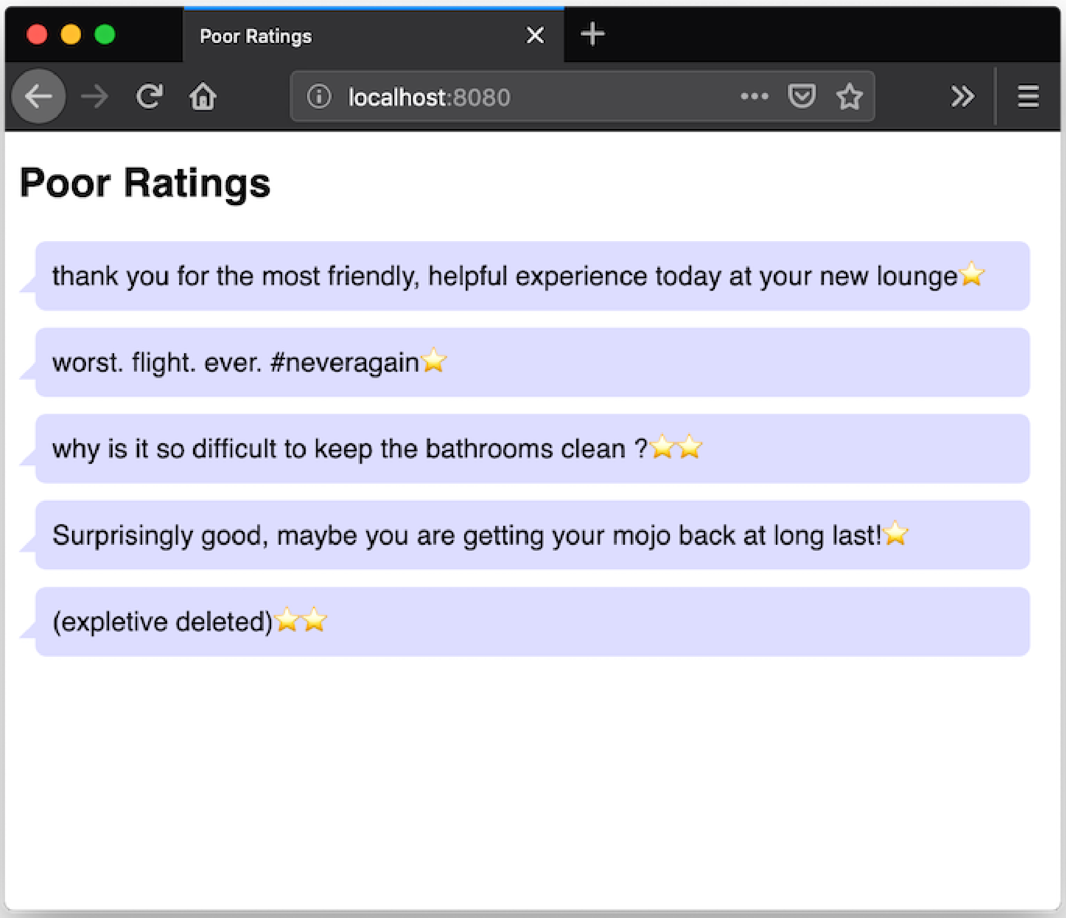
```
Local runtime statistics
------------------------
messages-per-sec:      0.33   total-messages:      1857     last-message:
 failed-messages:         0 failed-messages-per-sec:         0      last-f
(Statistics of the local KSQL server interaction with the Kafka topic POOR
```

N.B. you can use the up arrow on your keyboard to cycle through KSQL command history for easy access and replay of previous commands. Ctrl-R also works for searching command history.

6.2. Web Application for the derived stream

We now have the derived data in a Kafka topic. A team in our organization is interested in building an application to view this derived data in a web application.

A sample NodeJS websockets application has been built from the code in websockets. You can view the application at http://localhost:8080.



Poor Ratings

thank you for the most friendly, helpful experience today at your new lounge⭐

worst. flight. ever. #neveragain⭐

why is it so difficult to keep the bathrooms clean ?⭐⭐

Surprisingly good, maybe you are getting your mojo back at long last!⭐

(expletive deleted)⭐⭐

6.3. Query the stream

The derived stream that we've created is just another stream that we can interact with in KSQL as any other. If you run a `SELECT` against the stream you'll see new messages arriving based on those coming from the source `ratings` topic:

```
ksql> SELECT STARS, CHANNEL, MESSAGE FROM POOR_RATINGS;
1 | iOS | worst. flight. ever. #neveragain
2 | iOS | Surprisingly good, maybe you are getting your mojo back at long
2 | iOS | thank you for the most friendly, helpful experience today at you
```

Press Ctrl-C to cancel and return to the KSQL prompt.

6.4. It's just a Kafka topic…

The query that we created above (`CREATE STREAM POOR_RATINGS AS…`) populates a Kafka topic, which we can also access as a KSQL stream (as in the previous step). Let's inspect this topic now, using KSQL.

Observe that the topic exists:

```
ksql> SHOW TOPICS;

 Kafka Topic          | Registered | Partitions | Partition Replicas | Consu
--------------------------------------------------------------------------
[…]
 POOR_RATINGS         | true       | 4          | 1                  | 0
 ratings              | true       | 1          | 1                  | 1
--------------------------------------------------------------------------
ksql>
```

Inspect the Kafka topic's data

```
ksql> PRINT 'POOR_RATINGS';
Format:AVRO
6/20/18 11:01:03 AM UTC, 37, {"RATING_ID": 37, "USER_ID": 12, "STARS": 2,
6/20/18 11:01:07 AM UTC, 55, {"RATING_ID": 55, "USER_ID": 10, "STARS": 2,
```

6.5. Optional step: Alternative serialisation methods

By default KSQL will write to the target stream using the same serialisation as the source. So if it's reading Avro data, it'll write Avro data. You can override this behaviour using the `WITH (VALUE_FORMAT='xx')` syntax. For example, if for some terrible reason you wanted to write data to CSV in a topic, you would run:

```
CREATE STREAM POOR_RATINGS_CSV WITH (VALUE_FORMAT='DELIMITED') AS
  SELECT * FROM ratings WHERE STARS <3 AND CHANNEL='iOS';
```

You can use a similar syntax to change the name and partitions of the target Kafka topic:

```
CREATE STREAM POOR_RATINGS_JSON WITH (VALUE_FORMAT='JSON', PARTITIONS=2) A
  SELECT * FROM ratings WHERE STARS <3 AND CHANNEL='iOS';
```

Note	Re-serialising data this way can be a powerful use of KSQL in itself. Consider a source topic being written to in CSV format, and this topic is to be used by multiple consumers. One option is each consumer has to know the schema of the CSV. Another option is to declare the schema in KSQL once, and then reserialise the data into a better format such as Avro. Now each consumer can use the data without being tightly-coupled to the original system. *The SQL code would look like this - the source topic doesn't exist in your workshop environment so you won't be able to actually run it.*

```
CREATE STREAM USER_LOGON_CSV (first_name VARCHAR,
                              last_name  VARCHAR,
                              email      VARCHAR,
                              ip_address VARCHAR,
                              logon_date VARCHAR)
        WITH (KAFKA_TOPIC ='user_logons',
              VALUE_FORMAT='DELIMITED');

CREATE STREAM USER_LOGON WITH (VALUE_FORMAT='AVRO') AS
  SELECT * FROM USER_LOGON_CSV;
```

7. Kafka Connect / Integrating Kafka with a database

This shows how to ingest the Customers data from a database using Kafka Connect and CDC.

Check that Kafka Connect with Debezium's connector has started:

```
docker-compose logs -f connect-debezium|grep "Kafka Connect started"
```

Wait for the output:

```
connect-debezium_1  | 2018-09-04 11:33:04,639 INFO   ||  Kafka Connect sta
```

Press Ctrl-C to return to the command prompt.

We'll now create *two* Kafka Connect connectors. Both stream events from MySQL into Kafka using Debezium, but differ in how they handle the message structure.

```
docker-compose exec connect-debezium bash -c '/scripts/create-mysql-source
```

You should see `HTTP/1.1 201 Created` , twice.

Note	**Optional**: If you are interested you can inspect the script file (`scripts/create-mysql-source.sh` in the workshop folder) that includes the configuration payload in JSON. 1. The connector called `mysql-source-demo-customers` flattens the before/after record state data, along with the nested metadata, into a single flat structure. This is what we use during the rest of the workshop.

The flattening is done using a **Single Message Transform** from Debezium, called `io.debezium.transforms.UnwrapFromEnvelope` .

The connector also uses two Single Message Transforms to illustrate how metadata can be added to ingested data. The `InsertField` transformation adds the topic name into a field called `messagetopic` , and some fixed text into the `messagesource` field.

2. The connector `mysql-source-demo-customers-raw` retains the nested structure of the before/after record data.

   A Single Message Transform is used to route the messages to a different topic. By default Debezium will use the format `server.schema.table` when streaming a table's data to a Kafka topic. We use the `RegexRouter` to redirect the messages to a topic with a `-raw` suffix.

7.1. Exploring CDC change records

Start a MySQL command prompt:

```
docker-compose exec mysql bash -c 'mysql -u $MYSQL_USER -p$MYSQL_PASSWORD
```

Now in a separate terminal window run the following, to stream the contents of the customers topic and any changes to stdout:

```
# Make sure you run this from the `demo-scene/ksql-workshop` folder
docker-compose exec -T kafka-connect \
        kafka-avro-console-consumer \
        --bootstrap-server kafka:29092 \
        --property schema.registry.url=http://schema-registry:8081 \
        --topic asgard.demo.CUSTOMERS-raw --from-beginning|jq '.'
```

(jq is useful here—if you don't have it installed, remove `|jq '.'` from the above command).

Note the customer data shown, and the structure of it, with `before` , `after` , and `source` data.

From the MySQL command prompt, make some changes to the data:

```
INSERT INTO CUSTOMERS (ID,FIRST_NAME,LAST_NAME) VALUES (42,'Rick','Astley'
UPDATE CUSTOMERS SET FIRST_NAME = 'Thomas', LAST_NAME ='Smith' WHERE ID=2;
```

You should see each DML cause an almost-instantaneous update on the Kafka topic. For each change, inspect the output of the Kafka topic. Observe the difference between an `INSERT` and `UPDATE`.

7.2. Inspect a topic contents with KSQL - Customers

Let's look at the customer data from the KSQL prompt. This is pretty much the same as using `kafka-avro-console-consumer` as we did above.

Here we use the `FROM BEGINNING` argument, which tells KSQL to go back to the *beginning* of the topic and show all data from there

```
ksql> PRINT 'asgard.demo.CUSTOMERS' FROM BEGINNING;
Format:AVRO
3/4/19 5:50:42 PM UTC, Struct{id=1}, {"id": 1, "first_name": "Rica", "last
3/4/19 5:50:42 PM UTC, Struct{id=2}, {"id": 2, "first_name": "Ruthie", "la
```

Press Ctrl-C to cancel and return to the KSQL command prompt.

7.3. Re-Key Topics with KSQL

Since we're going to eventually join the customer data to the ratings, the customer Kafka messages *must be keyed on the field on which we are performing the join*. If this is not the case the join will fail and we'll get `NULL` values in the result.

Our source customer messages are currently keyed using the Primary Key of the source table, but using a key serialisation that KSQL does not yet support—and thus in effect is not useful as a key in KSQL at all.

To re-key a topic in Kafka we can use KSQL!

First we will register the customer topic.

```
ksql> CREATE STREAM CUSTOMERS_SRC WITH (KAFKA_TOPIC='asgard.demo.CUSTOMERS

 Message
----------------
 Stream created
----------------
ksql>
```

With the stream registered, we can now re-key the topic, using a KSQL `CSAS` and the `PARTITION BY` clause. Note that we're also changing the number of partitions from that of the source (4) to match that of the `ratings` topic (1):

Important	By changing the partition key, data may move between partitions,

and thus its ordering change. Kafka’s strict ordering guarantee only applies within a partition.

In our example this doesn’t matter, but be aware of this if you rely on this re-keying technique in other KSQL queries.

```
SET 'auto.offset.reset' = 'earliest';

CREATE STREAM CUSTOMERS_SRC_REKEY
        WITH (PARTITIONS=1) AS
        SELECT * FROM CUSTOMERS_SRC PARTITION BY ID;
```

```
 Message
----------------------------
 Stream created and running
----------------------------
ksql>
```

Note **Optional**

To inspect the key for a given stream/table, you can use the `ROWKEY` system column.

Here we compare it to the join column (`ID`); for the join to succeed they must be equal.

In the source stream, the `ROWKEY` reflects the key of the underlying Kafka messages as set by the producer (read by KSQL as a String):

```
ksql> SELECT C.ROWKEY, C.ID FROM CUSTOMERS_SRC C LIMIT 3;
 | 1
 | 2
 | 3
Limit Reached
Query terminated
```

In the re-keyed stream the `ROWKEY` and `ID` are equal, which is essential for a successful JOIN operation in KSQL.

```
ksql> SELECT C.ROWKEY, C.ID FROM CUSTOMERS_SRC_REKEY C LIMIT 3;
1 | 1
2 | 2
3 | 3
Limit Reached
```

```
Query terminated
ksql>
```

8. The Stream/Table duality

We're now going to model the Customers topic as a **KSQL Table**. This is a semantic construct that enables us to work with the data in the topic as key/value pairs, with a single value for each key. You can read more about this here.

Register a `TABLE` over the new re-keyed Kafka topic. Why's it a table? Because **for each key** (user id), we want to know **its current value** (name, status, etc)

```
ksql> CREATE TABLE CUSTOMERS WITH (KAFKA_TOPIC='CUSTOMERS_SRC_REKEY', VALU

 Message
---------------
 Table created
---------------
ksql>
```

Note *n.b. if you get the error* `Avro schema for message values on topic CUSTO` `Schema Registry.` *then* :

- Check that the topic's source stream is created:

```
ksql> SHOW STREAMS;
 Stream Name         | Kafka Topic         | Format
----------------------------------------------------
 CUSTOMERS_SRC_REKEY | CUSTOMERS_SRC_REKEY | AVRO
 [...]
```

- Verify that the source stream is processing messages by running `DESCF` Under the heading `Local runtime statistics` you should see:

```
Local runtime statistics
------------------------
messages-per-sec:      0.10   total-messages:        10     l
 failed-messages:         0 failed-messages-per-sec:
```

  - If no 'messages-per-sec' is shown, the next step is to verify that yo `'earliest';` earlier. You can run it again to be certain. If it says `Su` `'auto.offset.reset' from 'null' to 'earliest'` then the `null`
  - If this was the case, then you need to drop and recreate the stream

```
TERMINATE QUERY CSAS_CUSTOMERS_SRC_REKEY_0;
DROP STREAM CUSTOMERS_SRC_REKEY;
```

Then re-run the `CREATE STREAM CUSTOMERS_SRC_REKEY […]` from ab
queries running if the name differs from that shown in the `TERMINA`

Query the table:

```
ksql> SELECT ID, FIRST_NAME, LAST_NAME, EMAIL, CLUB_STATUS FROM CUSTOMERS
1 | Rica | Blaisdell | rblaisdell0@rambler.ru | bronze
3 | Mariejeanne | Cocci | mcocci2@techcrunch.com | bronze
4 | Hashim | Rumke | hrumke3@sohu.com | platinum
Limit Reached
Query terminated
```

Both `CUSTOMER` (table) and `CUSTOMER_SRC_REKEY` (stream) are driven from the same Kafka topic. You can examine the stream/table duality by running two KSQL sessions side-by-side, each querying one for a given key:

```
ksql> SET 'auto.offset.reset' = 'earliest';
Successfully changed local property 'auto.offset.reset' to 'earliest'. Use
ksql> SELECT ID, FIRST_NAME, LAST_NAME, EMAIL, CLUB_STATUS FROM CUSTOMERS
2 | Thomas | Smith | rbrockherst1@ow.ly | platinum
```

```
ksql> SET 'auto.offset.reset' = 'earliest';
Successfully changed local property 'auto.offset.reset' to 'earliest'. Use
ksql> SELECT ID, FIRST_NAME, LAST_NAME, EMAIL, CLUB_STATUS FROM CUSTOMERS_
2 | Ruthie | Brockherst | rbrockherst1@ow.ly | platinum
2 | Thomas | Smith | rbrockherst1@ow.ly | platinum
```

Leave each query running, and in a third window bring up a MySQL session

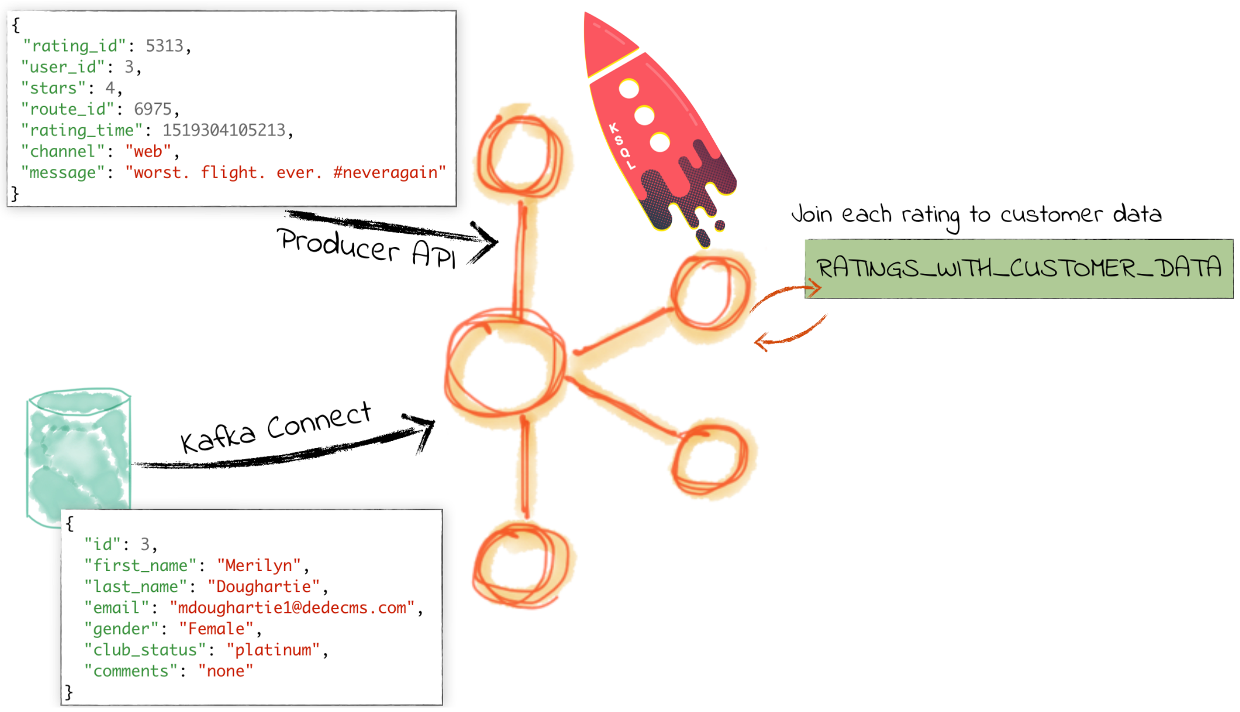
```
docker-compose exec mysql bash -c 'mysql -u $MYSQL_USER -p$MYSQL_PASSWORD
```

In MySQL make changes (one at a time) to the record for the ID being queried:

```
UPDATE CUSTOMERS SET EMAIL='foo@bar.com' WHERE ID=2;
UPDATE CUSTOMERS SET EMAIL='example@bork.bork.bork.com' WHERE ID=2;
```

When you make the change on MySQL you should see the two KSQL queries emit the new values almost immediately. Then, *cancel and re-run* the two KSQL queries (against table and stream respectively). Each time you should note that the stream contains **every** event related to the source table record, whilst the table hold the **current** state.

9. Joining Data in KSQL



```
{
 "rating_id": 5313,
 "user_id": 3,
 "stars": 4,
 "route_id": 6975,
 "rating_time": 1519304105213,
 "channel": "web",
 "message": "worst. flight. ever. #neveragain"
}
```

```
{
  "id": 3,
  "first_name": "Merilyn",
  "last_name": "Doughartie",
  "email": "mdoughartie1@dedecms.com",
  "gender": "Female",
  "club_status": "platinum",
  "comments": "none"
}
```

Let’s use the customer data (`CUSTOMERS`) and use it to enrich the inbound stream of ratings data (`RATINGS`) to show against each rating who the customer is, and their club status ('platinum','gold', etc).

Run the following SQL:

```
SELECT R.MESSAGE, C.FIRST_NAME, C.LAST_NAME
FROM RATINGS R INNER JOIN CUSTOMERS C
ON R.USER_ID = C.ID
LIMIT 5;
```

There are a couple of things to note about this query :

- We’re aliasing the table and stream names to make column names unambiguous
- The backspace character can be used to denote line continuation

In the output you should see a rating message, and the name of the customer who left it:

```
more peanuts please | Gianina | Mixhel
your team here rocks! | Munmro | Igounet
```

```
airport refurb looks great, will fly outta here more! | null | null
thank you for the most friendly, helpful experience today at your new loun
thank you for the most friendly, helpful experience today at your new loun
Limit Reached
Query terminated
ksql>
```

Now let's pull the full set of data, including a reformat of the timestamp into something human readable.

```
SELECT TIMESTAMPTOSTRING(R.RATING_TIME, 'yyyy-MM-dd HH:mm:ss'), R.RATING_I
R.MESSAGE, C.FIRST_NAME, C.LAST_NAME, C.CLUB_STATUS
FROM RATINGS R INNER JOIN CUSTOMERS C
ON R.USER_ID = C.ID;
```

```
2018-06-20 13:03:49 | 1 | 1 | 7562 | ios | more peanuts please | Gianina |
2018-06-20 13:03:50 | 2 | 4 | 54 | iOS | your team here rocks! | Munmro |
2018-06-20 13:03:51 | 4 | 1 | 7691 | web | thank you for the most friendly
2018-06-20 13:03:51 | 6 | 2 | 6902 | web | Surprisingly good, maybe you ar
```

Press Ctrl-C to cancel the output.

9.1. Populating a Kafka topic with the results of a Stream-Table join

Let's persist this as an enriched stream, by simply prefixing the query with `CREATE STREAM … AS`:

```
CREATE STREAM RATINGS_WITH_CUSTOMER_DATA WITH (PARTITIONS=1) AS
SELECT R.RATING_ID, R.CHANNEL, R.STARS, R.MESSAGE,
       C.ID, C.CLUB_STATUS, C.EMAIL,
       C.FIRST_NAME, C.LAST_NAME
FROM RATINGS R
     INNER JOIN CUSTOMERS C
       ON R.USER_ID = C.ID ;
```

```
 Message
----------------------------
 Stream created and running
----------------------------
```

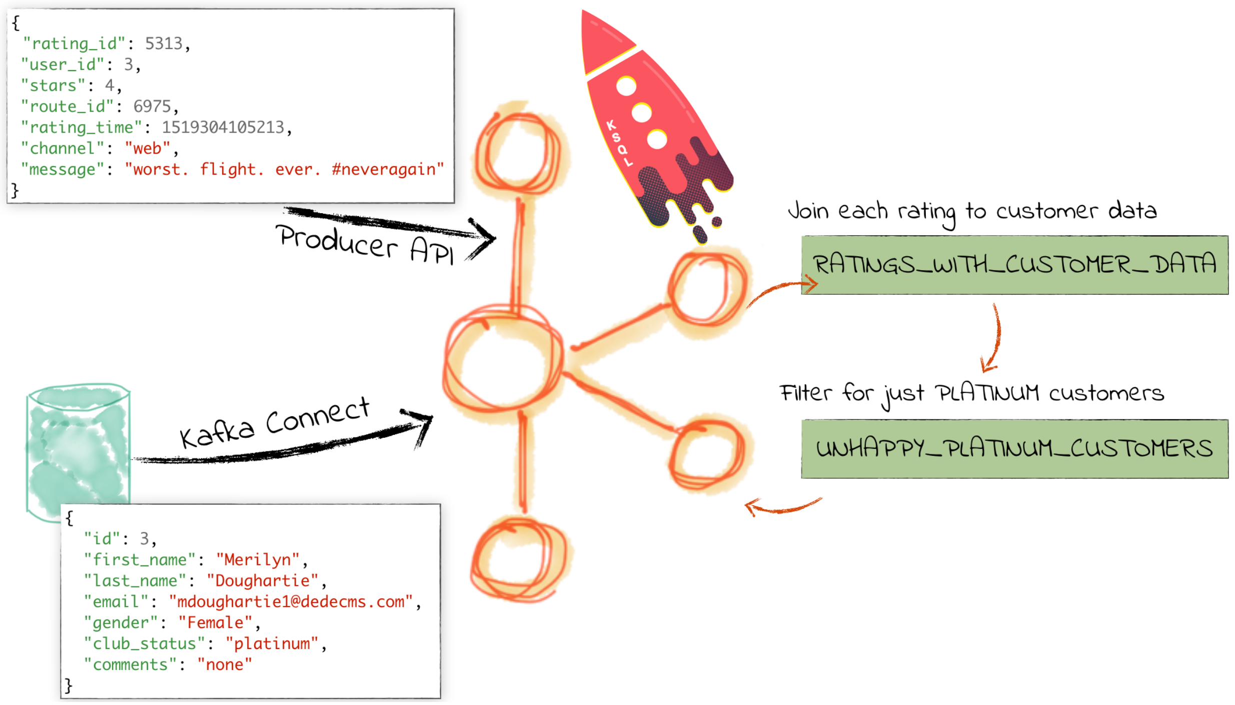
9.2. Filtering an enriched stream

Now that we have customer information added to every rating event, we can easily answer questions such as "Which of our Premier customers are not happy?":

```
SELECT EMAIL, STARS, MESSAGE
FROM RATINGS_WITH_CUSTOMER_DATA
WHERE CLUB_STATUS='platinum'
  AND STARS <3;
```

```
aarent0@cpanel.net | 2 | thank you for the most friendly, helpful experien
mdoughartie1@dedecms.com | 1 | worst.
flight.
ever. #neveragain
```

9.3. Daisy-chaining derived streams



```
{
 "rating_id": 5313,
 "user_id": 3,
 "stars": 4,
 "route_id": 6975,
 "rating_time": 1519304105213,
 "channel": "web",
 "message": "worst. flight. ever. #neveragain"
}
```

```
{
  "id": 3,
  "first_name": "Merilyn",
  "last_name": "Doughartie",
  "email": "mdoughartie1@dedecms.com",
  "gender": "Female",
  "club_status": "platinum",
  "comments": "none"
}
```

Producer API
KSQL
Join each rating to customer data
RATINGS_WITH_CUSTOMER_DATA
Filter for just PLATINUM customers
UNHAPPY_PLATINUM_CUSTOMERS
Kafka Connect

Having enriched the initial stream of ratings events with customer data, we can now persist a filtered version of that stream that includes a predicate to identify just those VIP customers who have left bad reviews:

```
CREATE STREAM UNHAPPY_PLATINUM_CUSTOMERS AS
SELECT CLUB_STATUS, EMAIL, STARS, MESSAGE
FROM   RATINGS_WITH_CUSTOMER_DATA
WHERE  STARS < 3
  AND  CLUB_STATUS = 'platinum';
```

```
 Message
----------------------------
 Stream created and running
----------------------------
ksql>
```

9.3.1. Query the new stream

Now we can query the derived stream to easily identify important customers who are not happy. Since this is backed by a Kafka topic being continually popuated by KSQL we can also drive other applications with this data, as well as land it to datastores down-stream for visualisation.

```
ksql> SELECT STARS, MESSAGE, EMAIL FROM UNHAPPY_PLATINUM_CUSTOMERS;
1 | is this as good as it gets? really ? | aarent0@cpanel.net
2 | airport refurb looks great, will fly outta here more! | aarent0@cpanel
2 | meh | aarent0@cpanel.net
```

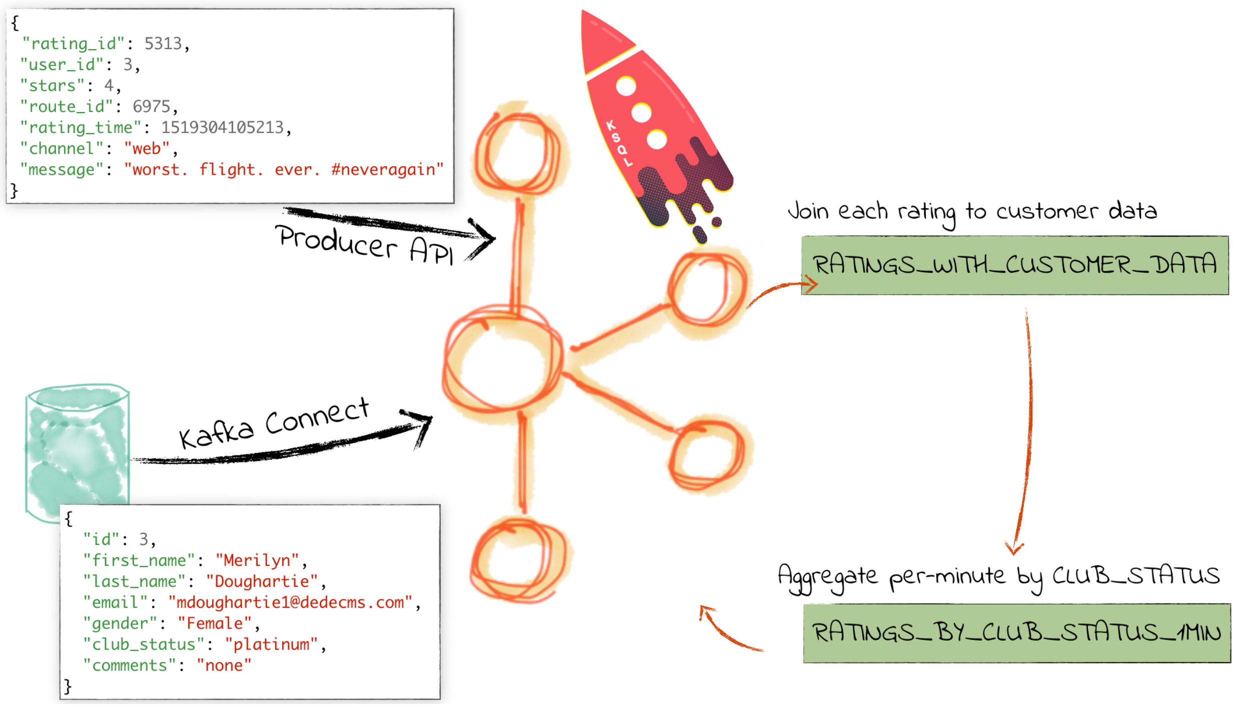
10. Streaming Aggregates

KSQL can create aggregations of event data, either over all events to date (and continuing to update with new data), or based on a time window. The time window types supported are:

- Tumbling (e.g. every 5 minutes : 00:00, 00:05, 00:10)
- Hopping (e.g. every 5 minutes, advancing 1 minute: 00:00-00:05, 00:01-00:06)
- Session (Sets a timeout for the given key, after which any new data is treated as a new session)

To understand more about these time windows, you can read the related Kafka Streams documentation. Since KSQL is built on Kafka Streams, the concepts are the same. The KSQL-specific documentation is also useful.

Note	KSQL will *re-emit* aggregates as they are updated by incoming events. This means that you will see multiple rows of output for a given window/key.


{
"rating_id": 5313,
"user_id": 3,
"stars": 4,
"route_id": 6975,
"rating_time": 1519304105213,
"channel": "web",
"message": "worst. flight. ever. #neveragain"
}
Producer API
KSQL
Join each rating to customer data
RATINGS_WITH_CUSTOMER_DATA
Kafka Connect
{
"id": 3,
"first_name": "Merilyn",
"last_name": "Doughartie",
"email": "mdoughartie1@dedecms.com",
"gender": "Female",
"club_status": "platinum",
"comments": "none"
}
Aggregate per-minute by CLUB_STATUS
RATINGS_BY_CLUB_STATUS_1MIN

10.1. Running Count per Minute

This shows the number of ratings per customer status, per minute:

```
SELECT TIMESTAMPTOSTRING(WindowStart(), 'yyyy-MM-dd HH:mm:ss'),
       CLUB_STATUS, COUNT(*) AS RATING_COUNT
FROM RATINGS_WITH_CUSTOMER_DATA
     WINDOW TUMBLING (SIZE 1 MINUTES)
GROUP BY CLUB_STATUS;
```

```
platinum | 1
bronze | 2
gold | 12
bronze | 13
```

The time window itself is exposed in the results using the function `WindowStart()`, which is then cast from epoch to human-readable form with the `TIMESTAMPTOSTRING` function.

Aggregates can be persisted too. Instead of `CREATE STREAM` as we did above, we're going to instead persist with a `CREATE TABLE`, since aggregates are always a table (key + value). Just as before though, a Kafka topic is continually populated with the results of the query:

```
CREATE TABLE RATINGS_BY_CLUB_STATUS AS
SELECT WindowStart() AS WINDOW_START_TS, CLUB_STATUS, COUNT(*) AS RATING_C
FROM RATINGS_WITH_CUSTOMER_DATA
     WINDOW TUMBLING (SIZE 1 MINUTES)
GROUP BY CLUB_STATUS;
```

```
 Message
---------------------------
 Table created and running
---------------------------
ksql>
```

This table that we've created is just a first class object in KSQL, updated in real time with the results from the aggregate query. Because it's just another object in KSQL, we can query and filter it as any other:

```
SELECT TIMESTAMPTOSTRING(WINDOW_START_TS, 'yyyy-MM-dd HH:mm:ss'),
        CLUB_STATUS, RATING_COUNT
FROM RATINGS_BY_CLUB_STATUS
WHERE CLUB_STATUS='bronze';
```

```
2019-03-04 17:42:00 | bronze | 2
2019-03-04 17:42:00 | bronze | 3
2019-03-04 17:42:00 | bronze | 5
2019-03-04 17:42:00 | bronze | 9
2019-03-04 17:42:00 | bronze | 10
```

If you let the `SELECT` output continue to run, you'll see all of the past time window aggregate values—but also the current one. Note that the *current* time window's aggregate value will continue to update, because new events are being continually processed and reflected in the value. If you were to send an event to the source `ratings` topic with a timestamp in the past, the corresponding time window's aggregate would be re-emitted.

11. Optional: Stream data to Elasticsearch

Note	This section assumes that you are familiar with the use of Kibana.

Using Kafka Connect you can stream data from a Kafka to one (or many) targets, including Elasticsearch, HDFS, S3, and so on.

Here we'll see how to stream it to Elasticsearch for rapid visualisation and analysis.

From a bash prompt, make sure that Elasticsearch and Kibana are running:

```
$ docker-compose ps|egrep "elasticsearch|kibana"
elasticsearch                      /usr/local/bin/docker-entr ...   Up
kibana                             /usr/local/bin/kibana-docker     Up
```

Create a dynamic mapping in Elasticsearch so that the timestamp of source data is correctly detected:

```
curl -XPUT "http://localhost:9200/_template/kafkaconnect/" -H 'Content-Typ
```

Create a connector to stream `RATINGS_WITH_CUSTOMER_DATA` to Elasticsearch:

```
curl -X "POST" "http://localhost:18083/connectors/" \
-H "Content-Type: application/json" \
-d '{
  "name": "es_sink_unhappy_platinum_customers",
  "config": {
    "connector.class": "io.confluent.connect.elasticsearch.ElasticsearchSi
    "topics": "RATINGS_WITH_CUSTOMER_DATA",
    "key.converter": "org.apache.kafka.connect.storage.StringConverter",
    "key.ignore": "true",
    "schema.ignore": "true",
    "type.name": "type.name=kafkaconnect",
    "topic.index.map": "RATINGS_WITH_CUSTOMER_DATA:ratings_with_customer_d
    "connection.url": "http://elasticsearch:9200",
    "transforms": "ExtractTimestamp",
    "transforms.ExtractTimestamp.type": "org.apache.kafka.connect.transfor
    "transforms.ExtractTimestamp.timestamp.field" : "TS"
  }
}'
```

Create a connector to stream `RATINGS_BY_CLUB_STATUS` to Elasticsearch:

```
curl -X "POST" "http://localhost:18083/connectors/" \
-H "Content-Type: application/json" \
-d '{
  "name": "es_sink_ratings_agg_by_status_1min",
  "config": {
    "connector.class": "io.confluent.connect.elasticsearch.ElasticsearchSi
    "topics": "RATINGS_BY_CLUB_STATUS",
    "key.converter": "org.apache.kafka.connect.storage.StringConverter",
    "key.ignore": "false",
    "schema.ignore": "true",
    "type.name": "type.name=kafkaconnect",
    "topic.index.map": "RATINGS_BY_CLUB_STATUS:ratings_agg_by_status_1min"
    "connection.url": "http://elasticsearch:9200"
  }
}'
```

Note that the above sets `"key.ignore": "false"` , and thus aggregates will be updated in-place.

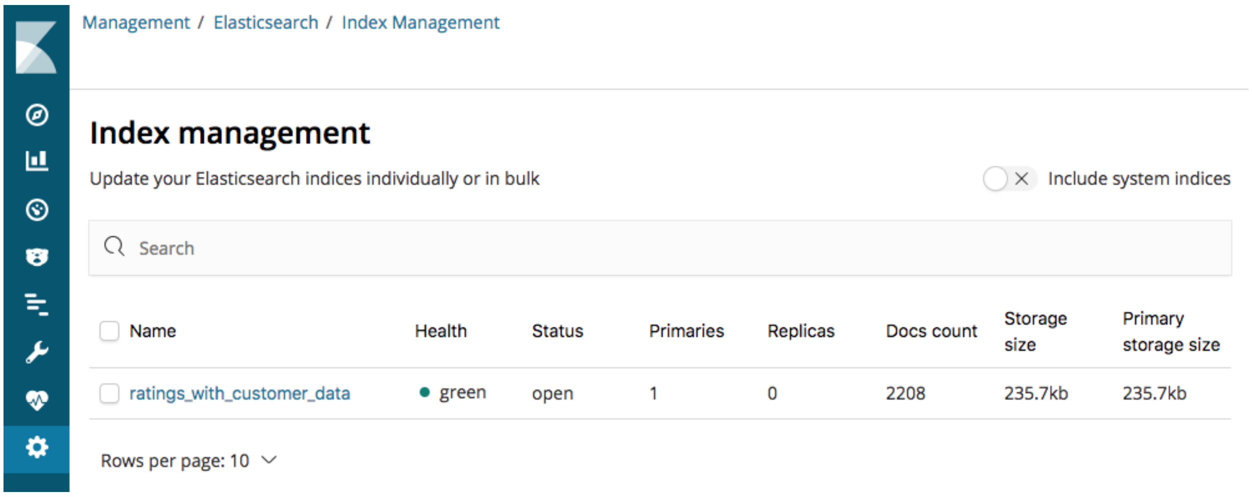
If you have `jq` on your machine you can run this to check that the connector is `RUNNING`:

```
$ curl -s "http://localhost:18083/connectors"| jq '.[]'| xargs -I{connecto

es_sink_ratings_agg_by_status_1min   |  RUNNING  |  RUNNING
es_sink_unhappy_platinum_customers   |  RUNNING  |  RUNNING
```

Use the Kibana interface to check that docs are arriving in Elasticsearch:

Note	If you get an error screen from Kibana then try restarting the container (`docker-compose restart kibana`).

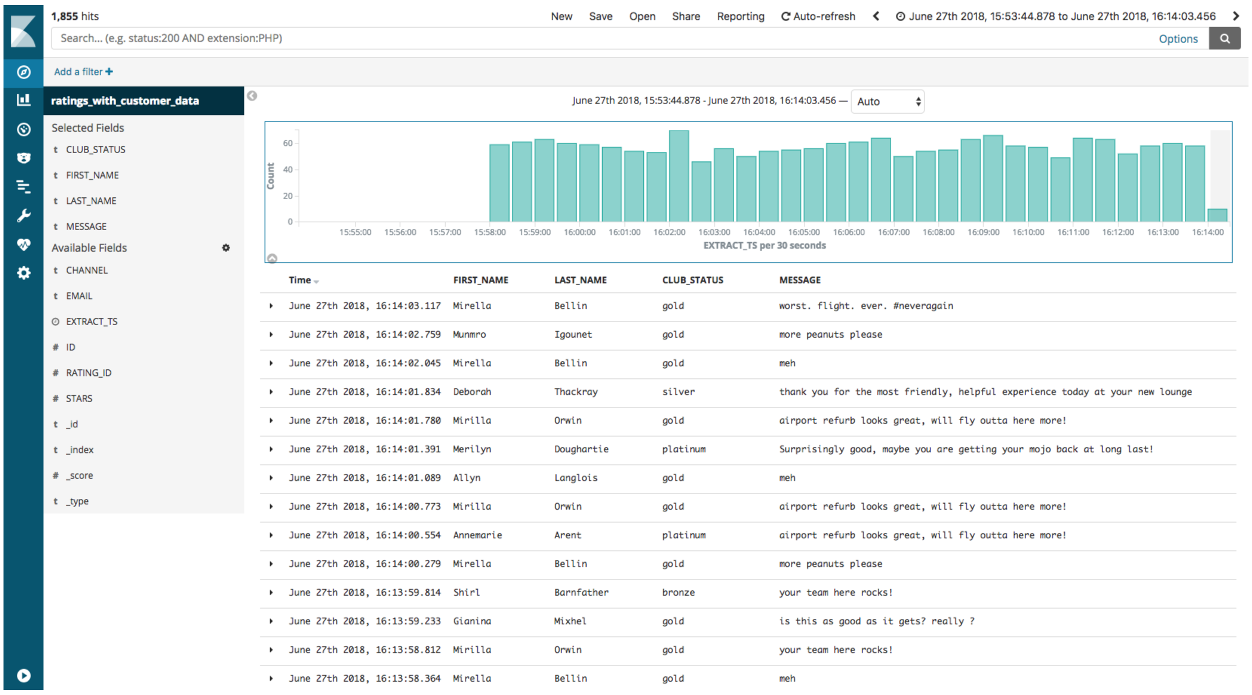


Index management

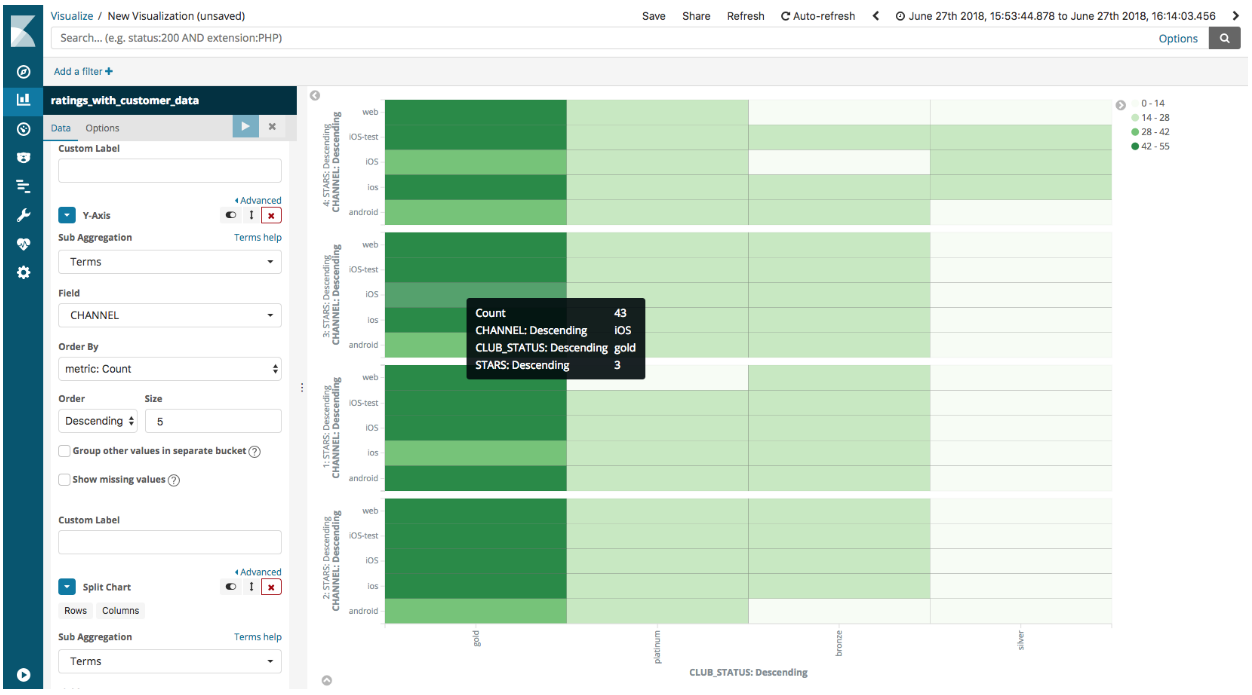
Update your Elasticsearch indices individually or in bulk

Name	Health	Status	Primaries	Replicas	Docs count	Storage size	Primary storage size
ratings_with_customer_data	● green	open	1	0	2208	235.7kb	235.7kb

Add the index pattern to Kibana, and then use the Discover and Visualise options to explore and create analyses on the data:


1,855 hits
New
Save
Open
Share
Reporting
Auto-refresh
June 27th 2018, 15:53:44.878 to June 27th 2018, 16:14:03.456
Search... (e.g. status:200 AND extension:PHP)
Options
Add a filter
ratings_with_customer_data
Selected Fields
CLUB_STATUS
FIRST_NAME
LAST_NAME
MESSAGE
Available Fields
CHANNEL
EMAIL
EXTRACT_TS
ID
RATING_ID
STARS
_id
_index
_score
_type
June 27th 2018, 15:53:44.878 - June 27th 2018, 16:14:03.456 —
Auto
Count
EXTRACT_TS per 30 seconds

Time	FIRST_NAME	LAST_NAME	CLUB_STATUS	MESSAGE
June 27th 2018, 16:14:03.117	Mirella	Bellin	gold	worst. flight. ever. #neveragain
June 27th 2018, 16:14:02.759	Munmro	Igounet	gold	more peanuts please
June 27th 2018, 16:14:02.045	Mirella	Bellin	gold	meh
June 27th 2018, 16:14:01.834	Deborah	Thackray	silver	thank you for the most friendly, helpful experience today at your new lounge
June 27th 2018, 16:14:01.780	Mirilla	Orwin	gold	airport refurb looks great, will fly outta here more!
June 27th 2018, 16:14:01.391	Merilyn	Doughartie	platinum	Surprisingly good, maybe you are getting your mojo back at long last!
June 27th 2018, 16:14:01.089	Allyn	Langlois	gold	meh
June 27th 2018, 16:14:00.773	Mirilla	Orwin	gold	airport refurb looks great, will fly outta here more!
June 27th 2018, 16:14:00.554	Annemarie	Arent	platinum	airport refurb looks great, will fly outta here more!
June 27th 2018, 16:14:00.279	Mirella	Bellin	gold	more peanuts please
June 27th 2018, 16:13:59.814	Shirl	Barnfather	bronze	your team here rocks!
June 27th 2018, 16:13:59.233	Gianina	Mixhel	gold	is this as good as it gets? really ?
June 27th 2018, 16:13:58.812	Mirilla	Orwin	gold	your team here rocks!
June 27th 2018, 16:13:58.364	Mirella	Bellin	gold	meh


Visualize / New Visualization (unsaved)

Search... (e.g. status:200 AND extension:PHP)

June 27th 2018, 15:53:44.878 to June 27th 2018, 16:14:03.456

ratings_with_customer_data

Y-Axis
Sub Aggregation: Terms
Field: CHANNEL
Order By: metric: Count
Order: Descending Size: 5
Group other values in separate bucket
Show missing values

Split Chart: Rows Columns
Sub Aggregation: Terms

Count 43
CHANNEL: Descending iOS
CLUB_STATUS: Descending gold
STARS: Descending 3

12. Shutting down the environment

To terminate the workshop environment, run `docker-compose down`:

```
$ docker-compose down
Stopping control-center    ... done
Stopping datagen-ratings   ... done
Stopping connect-debezium ... done
Stopping kafka-connect     ... done
Stopping ksql-cli          ... done
Stopping schema-registry   ... done
Stopping kafkacat          ... done
Stopping ksql-server       ... done
Stopping kibana            ... done
Stopping kafka             ... done
Stopping nginx             ... done
Stopping elasticsearch     ... done
Stopping websockets        ... done
Stopping zookeeper         ... done
Stopping mysql             ... done
Removing control-center    ... done
Removing datagen-ratings   ... done
Removing connect-debezium ... done
Removing kafka-connect     ... done
Removing ksql-cli          ... done
Removing schema-registry   ... done
Removing kafkacat          ... done
Removing ksql-server       ... done
Removing kibana            ... done
Removing kafka             ... done
Removing nginx             ... done
Removing elasticsearch     ... done
Removing websockets        ... done
Removing zookeeper        ... done
Removing mysql            ... done
Removing network ksql-workshop_default
```

If you want to preserve the state of all containers, run `docker-compose stop` *instead.*

13. Next steps

With the enriched and filtered data being populated into Kafka topics from KSQL you can use it to :

- Feed event-driven applications. For example, notify the ops team if a VIP user leaves a poor review.
- Stream to analytics platforms. For example, use Kafka Connect to stream the enriched data stream to Elasticsearch and visualise the real time with Kibana.